Evaluating Testing Frameworks for Ethereum Blockchain Applications: Introducing Wake

Josef Gattermayer, Ph.D.[1,2][0009−0008−0293−4209], Michal Převrátil[2], and Jan Kuběna[2]

[1] Czech Technical University in Prague, Faculty of Information Technology, Czech Republic https://fit.cvut.cz
[2] Ackee Blockchain, Czech Republic https://ackeeblockchain.com

Abstract. This paper introduces Wake, a new testing framework for Ethereum blockchain applications. We present a comparative study of Wake and established testing frameworks - Brownie, Ape, and Hardhat, using Uniswap v3 as a case study. The execution times of 271 test cases were measured across different combinations of development chains. Wake consistently outperformed the other testing frameworks, providing execution speeds 3-10 times faster than its closest competitor.

Keywords: Blockchain · Ethereum · Wake · Brownie · Ape · Hardhat · Anvil · Ganache · Uniswap v3.

1 Introduction

Blockchain is a relatively new technology enabling secure, decentralized, peer-to-peer online transactions. Ethereum[21] is a versatile blockchain platform facilitating the development of decentralized applications represented by smart contracts. These contracts, predominantly written in a higher-level language Solidity[6], adhere to the principle of "code is the law." This principle implies that, once deployed, the code dictates the contract's behavior, with no possibility of third-party intervention (unless mechanisms such as proxies[8] are introduced). Therefore, quality control of smart contract codes is a priority.

Within Ethereum and blockchain technology, *development chains* serve as testing environments. Development chains allow developers to deploy work-in-progress versions of their smart contracts while testing frameworks enable the creation of tests against these contracts. These tools are integral to smart contract development and testing, ensuring correct functionality and security before deployment onto the Ethereum network (Mainnet). Several testing frameworks, including Brownie[5], Ape[4], and Hardhat[17], are currently prevalent within the Ethereum development community.

This paper introduces Wake[2][16][15], a new testing framework for Ethereum blockchain applications. Our study aims to compare Wake with established testing frameworks, using Uniswap v3[3] as a case study. We measured the execution times of 271 Uniswap test cases using the testing frameworks in combination with various development chains. This study assesses Wake's efficiency and determines if it offers any performance advantages over other testing frameworks.

1.1 Ethereum

Ethereum operates on the Ethereum Virtual Machine (EVM), an environment that executes smart contracts. *Transactions* change the EVM's state. All transactions, including those deploying smart contract code, are immutable once confirmed, meaning they cannot be altered or undone.

An exception to this immutability is the use of design patterns like proxies, which introduce a level of upgradability to the contracts[8]. The EVM state is publicly readable, providing complete transparency of its present and historical state. The behavior is called "code is the law"—once a transaction, including smart contract interactions, is confirmed, it is permanent and cannot be reversed.

Testing on Ethereum is paramount due to unique platform attributes: immutability, transparency, and economic implications. These attributes differ distinctly from traditional software, where errors can be corrected post-deployment or malicious transactions can be reverted.

The *economic implications* underscore the necessity for comprehensive testing. Smart contracts frequently manage, transfer, or lock substantial value. Incorrect code or vulnerabilities can lead to significant financial losses. Recent attacks underscore these cost implications. One example in 2022 was the Nomad token bridge exploit[20], where approximately $190 million was drained due to a smart contract vulnerability failing to validate transaction inputs properly.

Uniswap v3 is a critical part of the Ethereum ecosystem, functioning as a decentralized trading protocol on the Ethereum blockchain. It facilitates direct, intermediary-free trading of Ethereum-based ERC20 tokens from user wallets.

Regarding its development infrastructure, Uniswap v3 is an open-source project with a comprehensive test suite written in TypeScript using the Hardhat testing framework.

As of May 2023, Uniswap v3 maintains considerable financial metrics: $2.85 billion in total locked value, an annualized volume of $506.8 billion, and annualized fees reaching $725.13 million[7]. Given its substantial scale and importance within the ecosystem, Uniswap v3 is an ideal case study for the comparative analysis of testing frameworks presented in this work.

2 Testing Frameworks

Testing frameworks are critical tools in software development, designed to support the creation and execution of test cases and the evaluation of test results. They provide a standardized environment where developers can test their software's functionality, performance, and reliability.

Ethereum testing frameworks provide functionalities like simulating transactions, interacting with smart contracts, checking contract states, and creating local testing blockchain environments (development chains).

This paper focuses on frameworks where tests are written in Python - Wake, Brownie, Ape. Results are compared with Hardhat, which provides test writing in TypeScript. Hardhat is a testing framework with a plugin Ethers.js[9]. The scope of this paper does not allow feature comparison of single frameworks as we focus only on the core functionality of running unit tests that support all the frameworks.

2.1 Development Chains

Development chains are pivotal components in the Ethereum development ecosystem. These local environments mimic the behavior of the Ethereum network, allowing developers to deploy and interact with smart contracts, send transactions, and check states, all within an isolated environment.

Development chains offer additional functionalities than public chains[14] (mainnets and testnets). These functionalities include sending unsigned transactions, changing the nonce of a given address, or temporarily suspending block mining. Furthermore, methods are available to obtain debugging information about individual transactions. These differences are useful for testing smart contracts due to their performance (test execution speed) and debugging capabilities.

The Anvil development chain is part of the Foundry[18] project. It is written in Rust, which can bring significant performance benefits. It implements all the necessary methods for debugging transactions, changing chain parameters, and operating with the chain.

Hardhat is a development chain written in Typescript. It implements almost the same set of methods as Anvil, making it very convenient to interact with.

Ganache is another development chain in the Ethereum ecosystem. Unlike Anvil and Hardhat, it offers fewer functionalities, affecting the functionalities offered by testing frameworks built on top of Ganache.

3 Wake Testing Framework

Wake is a new open-sourced Ethereum testing framework developed by Ackee Blockchain. This section presents the architecture, usage, and motivations behind the creation of Wake.

3.1 Motivation

The motivation behind the development of Wake is to create an all-in-one Python-based Swiss knife tool for smart contract development, testing, and auditing. The testing framework is the first part of this path. Wake provides an internal representation of smart contracts using Python's native data types.

Utilizing a Python-native data model in Wake offers significant advantages, mainly its inherent extensibility. Extended functionality offers Language Server

Protocol (LSP)[19] for Integrated Development Environments (IDEs)[19], fuzz testing[12] techniques, and test code coverage analysis[13].

Wake's priority is the speed of test execution. Wake's intended deployment is within Continuous Integration/Continuous Delivery (CI/CD)[11] pipelines, making it an ideal tool for projects with complex test suites.

3.2 Architecture

The Intermediate Representation (IR) model is at the heart of the Wake architecture, as it is constructed based on the Abstract Syntax Tree (AST) generated by the Solidity compiler, solc[3]. This model conforms to the AST's node types and structural elements and is crucial for several components of Wake, including the language server, testing framework, and static analysis[10].

The IR model merges the outputs of the compiler into a single model, which allows for a high degree of parallelization in the compilation process. Additionally, the model enables incremental compilation, where only modified files must be recompiled, and the results are integrated into the unified model. The IR model also corrects errors and inconsistencies in the AST in different compiler versions, ensuring consistency and reliability.

The IR model is stored in a binary file format using the standard Python module Pickle[4], which allows fast loading from disk.

After compilation and creation of the IR model, the Wake testing framework generates *pytypes*, which are Python files containing the definitions of contracts, enums, and structs in native Python objects such as classes, data classes, or IntEnums. These pytypes files are divided into directories/modules corresponding to Solidity's original source files' directory hierarchy. The framework generates methods corresponding to the public interface of the given contract, allowing for easy testing and interaction with the contracts on the chain.

Using pytypes provides several testing benefits, including auto-completion and type-checking when writing tests. Additionally, pytypes allow deploying contracts and calling their methods, making testing and interacting with contracts easy to use.

3.3 Usage

Interaction with contracts in the Wake framework is accomplished through low-level methods or the methods generated in 'pytypes'. Four low-level methods represent different requests: 'tx', 'call', 'estimate', and 'access_list'. The '.transact()', '.call()', '.estimate()', and '.access_list()' methods correspond to these request types, respectively.

The 'pytypes'-generated methods default to 'tx' for non-pure non-view functions and 'call' for pure and view functions, but the request type can be modified.

[3] https://docs.soliditylang.org/en/latest/installing-solidity.html
[4] https://docs.python.org/3/library/pickle.html

Low-level methods and ‘pytypes‘-generated methods accept common keyword arguments to all request types. These arguments can be used to modify specific parameters of transactions and requests sent in the testing framework, such as the sender address (‘from_‘), the value sent along with the transaction (‘value‘), the gas limit (‘gas_limit‘), the gas price (‘gas_price‘), the maximum fee per gas (‘max_fee_per_gas‘), the maximum priority fee per gas (‘max_priority_fee_per_gas‘), the access list (‘access_list‘), and the request type (‘type‘).

After a transaction is sent, the testing framework returns a transaction object that provides access to attributes such as call traces, events, and console logs, even in the case of a transaction failure. The framework offers advanced features, such as the ability to attach a debugger in the case of a transaction revert. The debugger allows developers to inspect Python objects within the script and contracts on the chain by invoking ‘pytypes‘ or low-level functions. The framework also provides helper functions for ABI[5] encoding and decoding, creating and restoring chain snapshots, and native support for cross-chain testing without context switching using context blocks. Furthermore, the framework allows Solidity code coverage analysis of selected test scripts, although performance may be reduced.

4 Methodology

Performance evaluation was chosen as the primary metric for this study. A testing framework's performance can indicate its internal design efficiency and influence its adoption by developers and integration in CI/CD pipelines. Slow testing frameworks limit users, potentially leading to test skips or size reduction, undermining blockchain safety and stability.

4.1 Test Suite

A subset of the original Uniswap v3 Hardhat TypeScript tests was rewritten into Python, then further adapted to be compatible with each Python framework under consideration. The adapted test suite included 271 test cases and parameterized tests [1].

Adjustments[6] to the Brownie framework were required for its compatibility with Anvil, as Brownie does not natively support the `--block-base-fee-per-gas` option. Without this option, tests would fail. To avoid Anvil logging approximately 15 GB of data per test, we added the `prune-history 100` option.

4.2 Measurement Process

The test execution process was automated to ensure consistency[1]. Initially, a framework-specific compile command was executed, followed by a single 'dummy'

[5] https://docs.soliditylang.org/en/develop/abi-spec.html
[6] https://github.com/Ackee-Blockchain/Wake-measurements/blob/brownie/network/rpc/anvil.py

test run not included in the recorded execution times to mitigate potential caching effects.

Each framework and development chain combination was run ten times consecutively. The average execution time was calculated. The execution time was determined by the `time` command as the 'real' output time.

All Python frameworks were set up within virtual environments using packages specified in their requirements files. For Hardhat, the required packages were installed from the `packages.json` file.

Tests were conducted on a VM n2d-standard-4 instance in Google Cloud, equipped with four cores of AMD Rome, 16 GB RAM, and an HDD disk. The system ran a Debian OS with kernel version `5.10.162-1` and Python 3.10.11. All tests were executed under similar load conditions on the virtual machine.

5 Results and Discussion

The data collected from test suite executions are displayed in *Table 1.* It shows Wake framework's leading performance, being 3.3x faster on the Anvil development chain than any other framework combination. Tests were executed and measured 200 times.

Table 1. Average execution times of the test suite for different testing frameworks (columns) and development chains (rows). Results in seconds are shown as: mean (standard deviation)

	Brownie [s]	Ape [s]	Wake [s]	Hardhat Ethers.js [s]
Anvil	34.80 (1.31)	53.27 (1.22)	**3.37 (0.05)**	10.96 (0.40)
Ganache	51.48 (1.41)	72.72 (2.09)	**15.78 (0.22)**	118.71 (1.69)
Hardhat	51.62 (2.43)	72.42 (1.80)	19.69 (0.15)	**17.47 (0.17)**

6 Conclusion

The present study introduced Wake, a new testing framework featuring a Python-native data model and Intermediate Representation architecture. Through performance measurements against established frameworks Brownie, Ape, and Hardhat with Ethers.js on different development chains (Anvil, Ganache, Hardhat), Wake demonstrated performance gains in executing test suites, mainly when used with Anvil. Key performance attributes of Wake, such as high degrees of parallelization during compilation and incremental compilation, significantly contributed to these efficiencies.

Our findings also highlight the significant impact of development chains on the performance of a testing framework. Notably, Ganache consistently underperformed relative to Anvil and Hardhat across all testing frameworks evaluated.

Future work will leverage Wake's intermediate representation model architecture to incorporate additional features, including a static analysis, fuzz testing, and a test coverage visualization.

References

1. Ackee Blockchain: Github repository, https://github.com/Ackee-Blockchain/python-testing-frameworks-benchmark/blob/master/test/_projects.py, Last accessed 2023-05-19
2. Ackee Blockchain: Wake documentation, https://ackeeblockchain.com/wake/docs/latest/, Last accessed 2023-05-19
3. Adams, H., Zinsmeister, N., Salem, M., Keefer, R., Robinson, D.: Uniswap v3 core. Tech. rep., Uniswap, Tech. Rep. (2021)
4. ApeWorx LTD: Ape homepage, https://www.apeworx.io/, Last accessed 2023-05-19
5. Brownie: https://eth-brownie.readthedocs.io/en/stable/, Last accessed 2023-05-19
6. Dannen, C.: Introducing Ethereum and solidity, vol. 1. Springer (2017)
7. DefiLlama: https://defillama.com/protocol/uniswap-v3, Last accessed 2023-05-19
8. Ethereum Foundation: https://ethereum.org/en/developers/docs/smart-contracts/upgrading/#proxy-patterns, Last accessed 2023-05-19
9. Ethers: Ether.js documentation, https://docs.ethers.org/v5/, Last accessed 2023-05-19
10. Feist, J., Grieco, G., Groce, A.: Slither: a static analysis framework for smart contracts. In: 2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB). pp. 8–15. IEEE (2019)
11. Fowler, M., Foemmel, M.: Continuous integration (2006)
12. Grieco, G., Song, W., Cygan, A., Feist, J., Groce, A.: Echidna: effective, usable, and fast fuzzing for smart contracts. In: Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 557–560 (2020)
13. Ivanković, M., Petrović, G., Just, R., Fraser, G.: Code coverage at google. In: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 955–963 (2019)
14. Iyer, K., Dannen, C., Iyer, K., Dannen, C.: The ethereum development environment. Building Games with Ethereum Smart Contracts: Intermediate Projects for Solidity Developers pp. 19–36 (2018)
15. Jan, K.: Rozšíření nástroje Woke. Master's thesis, České vysoké učení technické v Praze. Vypočetní a informační centrum. (2023)
16. Michal, P.: Rozšíření nástroje Woke. Master's thesis, České vysoké učení technické v Praze. Vypočetní a informační centrum. (2022)
17. Nomic Foundation: Hardhat homepage, https://hardhat.org/, Last accessed 2023-05-19
18. Paradigm: Foundry homepage, https://getfoundry.sh/, Last accessed 2023-05-19
19. Rodriguez-Echeverria, R., Izquierdo, J.L.C., Wimmer, M., Cabot, J.: Towards a language server protocol infrastructure for graphical modeling. In: Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems. pp. 370–380 (2018)
20. Scharfman, J.: Decentralized finance (defi) fraud and hacks: Part 2. In: The Cryptocurrency and Digital Asset Fraud Casebook, pp. 97–110. Springer (2023)
21. Wood, G., et al.: Ethereum: A secure decentralised generalised transaction ledger. Ethereum project yellow paper **151**(2014), 1–32 (2014)